ORACLE

ORACLE

Scalable Application Design

Bjørn Engsig
Principal Member of Technical Staff
Oracle Server Technologies

Agenda

- Who am I?
- Background
- The scalable database software
- Cursor processing
- Session pooling
- Connection pooling
- Summary

Who am I?

- Bjørn Engsig
- Real World Performance Team
- Wrote several versions of the white paper "*Designing Applications for Performance and Scalability*"
- Almost famous for saying "It's the applications fault"

Background

- Scalability comes from a balance between private and shared information

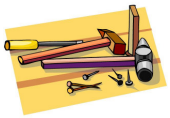
Shared	*Private*
Server	Your PC
Database instance	Why not your own?
Buffer cache	Sort area
SQL parse tree and execution plan	Your bind values and results
Main memory	L1 and L2 cache

Background – resource sharing

- Effective resource usage
- Need for a protocol
  - Latches
  - Mutex
  - Cache coherency
  - Cooperative clients
- Important metrics
  - Cost of acquiring/releasing a resource
  - Time spent using the resource

Two carpenters with one toolbox





- Who uses the tools when?

This presentation

- SQL statements/cursors
  - Parsing and execution
  - Bind variables
  - Pitfalls and quirks
- Database sessions
  - Sharable or not
  - Session pools
  - Application requirements, multithreading

This presentation ...

- SQL*Net Connections
  - Connection pooling
  - Application possibilities
- Database Resident Connection Pool
- Shared Server
- Summary
  - When to use what

The scalable database software

- Server side
  - Ability to share cursor information
  - Soft parsing
  - Hard parsing
  - SQL literal handling
  - Library cache
- Client side
  - New design from scratch (OCI in version 8)
  - Separation of connection, session and transaction
  - Pools everywhere
- Very early design decisions proved very correct

The scalable database software

Efficient use of bind variables, cursor_sharing and related cursor parameters

An Oracle White Paper
August 2001

Designing applications for performance and scalability

An Oracle White Paper
July 2005

Cursor processing

- A ”cursor” is a handle to the execution of a particular SQL statement (or PL/SQL block)
- Associates the client side execution with runtime data to the server side
- Server side split between shared and private
- Apparently same representation on client and server
- Quite subtle and quite important differences, however
- Client APIs are different

Cursor processing – server side

- Parse
  - Initial (hard) parse primarily does syntax checking and object existence
  - Subsequent (soft) parse verifies object availability, access control, etc
- Optimize
  - Mostly done during first execute
  - Never explicitly done
- Execute
  - Actually uses runtime specific data (queries, DML, PL/SQL)
- Fetch
  - Returns results for queries only

ORACLE

Cursor processing – client (or application server) side

- Parse/prepare
  - Prepares a SQL statement for execution
- Bind
  - Links placeholders in SQL statement to application data
- Execute
  - Not intuitive what actually happens on the server
  - Different API's behave differently
- Define
  - Defines application storage for select list elements
- Fetch
  - Retrieves query results into defined storage

ORACLE

Cursor processing – sample code

```
cursor cur;
number custid, ordid;
parse(cur,
  "insert into orders(custid, ordid) values (:1, :2)");
loop
  custid := /* some value */
  ordid := /* some value */
  bind(cur, 1, custid);
  bind(cur, 2, ordid);
  execute(cur);
  commit;
end loop;
```

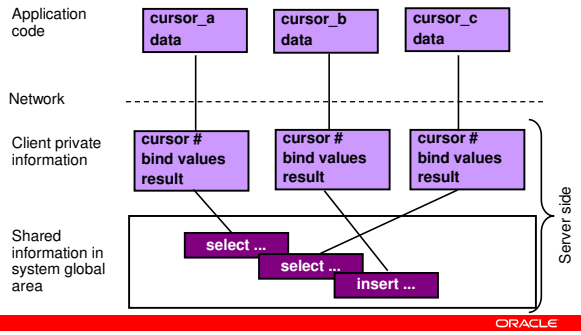
ORACLE

Cursor processing – sample code with soft parse

```
loop
  cursor cur;
  number custid, ordid;
  parse(cur,
    "insert into orders(custid, ordid) values (:1, :2)");
  custid := /* some value */
  ordid := /* some value */
  bind(cur, 1, custid);
  bind(cur, 2, ordid);
  execute(cur);
  commit;
end loop;
```

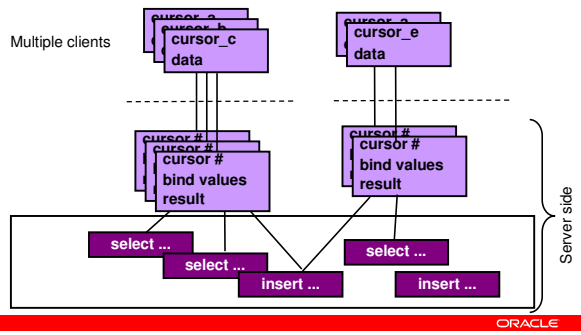
ORACLE

Cursor processing – application and database server



ORACLE

Cursor processing – application and database server



ORACLE

Cursor processing – frequent execution

- Remember the carpenter – cost of acquiring (low to moderate) vs. cost of using (low)
- Keep cursor open on client side
- Avoids the soft parse
- Just change bind values between execution
- Make sure different clients use the same SQL statement
- Beware of bind value peeking

- Still the best despite recent library cache changes

ORACLE

Cursor processing – infrequent execution

- Remember the carpenter – cost of acquiring (moderate) vs. cost of using (high)
- Using literals make the optimizer do its job well
- You want the statement optimized for each execution
  - Also with different clients
- Beware of bind value peeking

ORACLE

Cursor processing – sample code with hard parse

```
cursor cur;
parse(cur,
    "select /* something complex */
      from  /* many tables */
      where /* join predicates */
      and t1.a=42
      and t2.b=2
      and (t3.c=4 or t3.d like 'ABC%')
      and t4.e in (7,9,13)");
define(cur, ... );
execute(cur);
```

ORACLE

Cursor processing – common mistakes

- Bind peeking fixing execution plan
  - Different plans really needed for different values

```
cursor cur;
number custid, status;
parse(cur,
  "select * from orders where custid=:1 and status =:2)");
loop
  custid := /* some value */
  status := /* some value */
  bind(cur, 1, custid);
  bind(cur, 2, status);
  execute(cur);
  define(cur, ...);
  fetch;
end loop;
```

ORACLE

Cursor processing – common mistakes

- Using SQL to do procedural logic

```
cursor cur;
number custid
string custname;
parse(cur,
  "select * from orders where
      (:1 is not null and custid=:1)
   or (:2 is not null and custname=:2)");
loop
  custid :=   /* some value - maybe NULL */
  custname := /* some value - maybe NULL */
  bind(cur, 1, custid);
  bind(cur, 2, custname);
  execute(cur);
  define(cur, ...);
  fetch(cur);
end loop;
```

ORACLE

Cursor processing – why these mistakes?

- Application programmers have finally learned using bind variables
- "avoid the soft parse" mantra leads to certain programming style
- Hard to keep track of many SQL statements with only minor differences

ORACLE

Cursor processing – and the solution

- Let API handle cursor re-use using a cursor cache
- Application does prepare/release in stead of parse
- Size of cursor cache should be properly controlled
  - Too high implies memory wastage
  - Too low implies soft parsing

ORACLE

Cursor processing – and the solution

```
loop
  cursor cur;
  number custid
  string custname;
  if /* custid is known */
    prepare(cur, "select * from orders where custid=:1)
    bind(cur, 1, custid);
  else /* find using name */
    prepare(cur, "select * from orders where custname=:1)
    bind(cur, 1, custname);
  end if;
  execute(cur);
  define(cur, ...);
  fetch(cur);
  release(cur);
end loop;
```

With prepare/release pair, the API will automatically cache the SQL statements ready for execution

ORACLE

Cursor processing – bind peeking in Oracle11g

- Bind peeking at first execution stores optimizer information
- At subsequent executes, execute layer realizes bind value has changed sufficiently to make execution plan invalid
- New execution plan created and stored
- Several concurrent execution plans
- You may see higher library cache demand

ORACLE

SQL*Net roundtrips

- Roundtrip between application and database server have cost
  - Involve TCP/IP stack and physical network
  - SQL*Net is very chatty
- Reducing number of roundtrips increases network performance and scalability
- Always use API's feature for roundtrip reduction
  - The execute call is often a "do-it-all" call, that does parse (if needed), bind and server side execute
  - Array interface
  - Mostly done implicitly

ORACLE

SQL*Net – array fetch

- Automatically done in most API's
- The "do-it-all" execute call will also implicitly fetch a number of rows
- Controlled via row count or memory size
  - Don't forget to set it!
- Beware of older API's
- Can have high impact

ORACLE

SQL*Net – array fetch

```
cursor cur;
number custid, ordid, status;
parse(cur,
  "select custid,ordid from orders where status =:1)");
status := /* some value */
bind(cur, 1, status);
execute(cur);
define(cur, 1, custid);
define(cur, 2, ordid);
while fetch(cur) loop
  /* process one row */
end loop;
```

first actual roundtrip done here, also fills implicit array with rows

incurs roundtrip only when array is exhausted

ORACLE

SQL*Net – array DML

- Mostly used for insert operations
- Most API's require actual application implementation using array
- Big impact on massive inserts

SQL*Net – array DML

Actual arrays declared

```
cursor cur;
number custid[N], ordid[N]; /* declare arrays */
parse(cur,
  "insert into orders(custid, ordid) values (:1, :2)");
for i in 1..N loop
  custid[i] := /* some value */
  ordid[i] := /* some value */
end loop;
bindarray(cur, 1, custid);
bindarray(cur, 2, ordid);
executearray(cur, N);
commit;
```

Bind explicitly to arrays

Complete array processed in one roundtrip

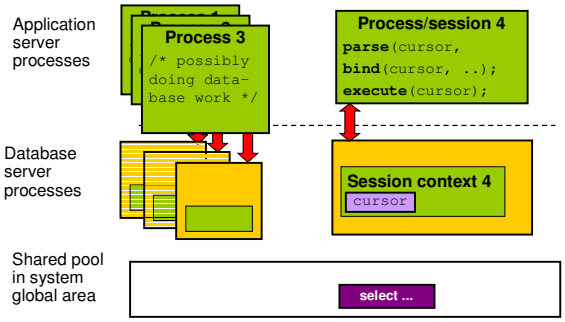
Application/server connections

- Traditional dedicated server
- Session pooling
- Connection pooling
- Database Resident Connection Pool
- Shared Server
- Comparison

Traditional Dedicated Server

- Connection and session established at the same time
- Kept until logoff
- Single or multithreaded application
- Hard upper limit
  - Thousands possible
  - Hundreds of thousands are not possible

Dedicated server – traditional client/server



Session pooling – overview

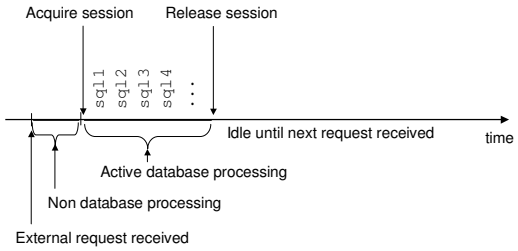
- A *session* is the context within which a specific (application) user executes SQL statements
  - Cursors are allocated under sessions
- High cost of creating a session (compared to execution of a SQL statement)
- Sessions are a limited resource
  - Thousands are possible
  - Hundreds of thousands are not
- High memory requirements on database server
  - Mostly due to many open cursors
- Sessions may contain state
  - NLS environment, PL/SQL package data, etc.
  - State information may make sharing or reuse difficult

Session pooling

- Multithreaded application
- Many threads taking user request
- Pooled sessions actually do database work

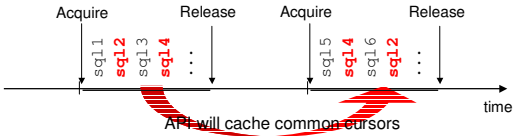
- Application requirements:
  - Multithreaded
  - Code must explicitly acquire and release sessions
  - No state kept after session release
  - Efficient sharing requires homogeneity between sessions
    - same username, NLS environments, other session setting
  - Control of request count and session count

Session pooling – example



Session pooling

- "No state kept" used to imply, no open cursors kept
- Statement cache and cursor prepare/release will overcome this
- Always use newer calls
- Full scalability ensured



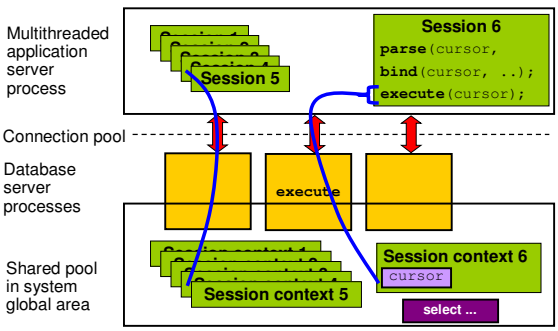
Connection pooling – overview

- A *connection* is the SQL*Net transport protocol from the application to the database server
  - Sessions run on top of connections
- High cost creating a connection
  - Network stack on application and database server
  - Process creation by the O/S
- Connections are a limited resource
  - Thousands are possible
  - Hundreds of thousands are not
- Connections contain no state information

Connection pooling – overview

- Multithreaded applications
- Completely homogeneous and re-usable connections
- A connection is automatically picked for each SQL*Net roundtrip
- Very few application changes
  - No need to (explicitly) acquire/release connections
  - Sessions can keep state for a long time
- Session memory allocated on database server
  - Different allocation scheme than with dedicated connections
  - Watch out for memory overhead when sessions have long idle times
- Control of connection pool required

Connection pooling

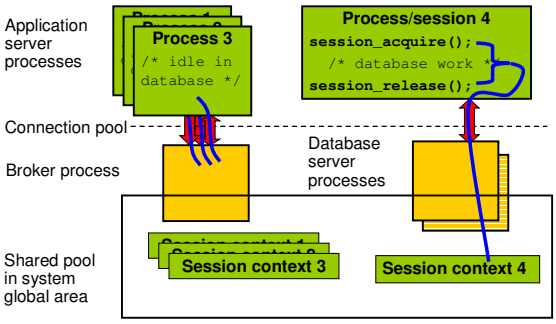


Database Resident Connection Pool

- New feature in Oracle11g
- Allows pooling between application processes
- Multi threaded application not needed
- Actually provides a session pool
  - Session acquire/release needed
- Uses a broker on the database server side

ORACLE

Database Resident Connection Pool



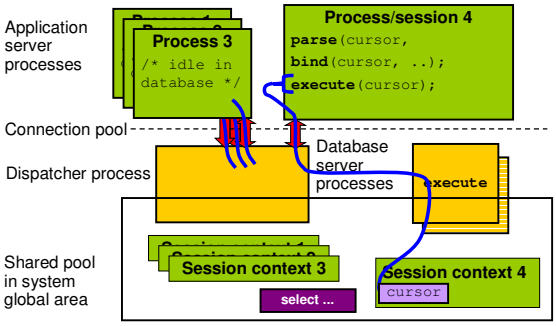
ORACLE

Shared Server

- Pooling purely on the database server side
- Completely transparent to application code
- All requests go via a dispatcher
  - Performance overhead
  - Process switch

ORACLE

Shared Server



ORACLE

Pooling comparison

• Method	• Connection or pooling method
• Coding	• Application coding requirements
• Server	• Administration complexity on server side
• Scalability	• Possibilities for scalable application design
• Performance	• Raw performance implications not including scalability
• Usability	• Typical application types using this method

ORACLE

Direct, dedicated connection

- Traditional stateful or stateless application coding, complete session and cursor control
- Simple server administration
- Scalability can be fully ensured using proper cursor control avoiding soft parsing. Hard upper limit on concurrent connections.
- Best possible performance
- Recommended and very efficient for batch processing, in particular with client imposed control on number of concurrent batch processes.

ORACLE

Connection pooling

- Stateful or stateless application coding, must be multithreaded
- Complex server configuration as private session memory is in SGA and uneven process and session count.
- Full scalability ensured with proper cursor processing, concurrency limitation imposed by application server
- Best possible performance
- Recommended for back-office type applications with relatively constant think-time and processing time, works well with rich session state (e.g. PL/SQL packages variables, long running cursors)

ORACLE

Session pooling

- Stateless applications only, sessions must explicitly be acquired and released, must be multi threaded
- Relatively simple server configuration
- Scalability possible with homogeneous sessions, use of prepare/release API is necessary
- Possibly very good performance, homogeneous sessions required
- Recommended for Internet type applications and for back-office type applications that can use homogeneous sessions, not suited for batch processing

ORACLE

Database Resident Connection Pool

- Stateless applications only, sessions must explicitly be acquired and released, can be single or multi threaded
- Complex server administration, session private memory in SGA, DRCP configuration
- Scalability via server side, rarely possible to avoid soft parse
- Good performance, little overhead with session acquire/release
- Recommended for Internet type applications that cannot run multi-threaded such as PHP, not suited for batch processing

ORACLE

Shared Server

- Any application type will work
- Complex server configuration of dispatchers, servers and session private memory in SGA
- Minor scalability impact of dispatchers
- Performance consistently impacted by dispatchers
- Rarely good, but the only possibility for stateful, single threaded applications that have long idle times

ORACLE

Summary

- Oracle improves scalability with each version
  - We see so many real world cases; we know where the typical problems are.
- Some improvements come for free
  - Benefit for new and existing applications
- Some improvements require application implementation
- Good application design and development continues to be important
- Added features add to decision complexity

ORACLE

Q & A

ORACLE


ORACLE®